Declarative Programming of Interactive Systems with Grapefruit

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Software Technology Colloquium at Utrecht Universiteit
May 29, 2008
Grapefruit overview

- Haskell library for Functional Reactive Programming with a focus on GUIs and animated graphics
- push-based implementation for efficiency
- based on Gtk2Hs and HOpenGL, multi-toolkit support planned
- currently developed at the BTU Cottbus
- several Cabal packages:
  - *grapefruit-frp* Functional Reactive Programming core
  - *grapefruit-records* record system
  - *grapefruit-gui* GUI programming
  - *grapefruit-graphics* graphics programming
  - *grapefruit-examples* illustrations of Grapefruit’s features
- homepage at
  - [http://haskell.org/haskellwiki/Grapefruit](http://haskell.org/haskellwiki/Grapefruit)
This talk

- outline:
  1. Functional Reactive Programming
  2. Record system
  3. Multi-toolkit support

- everything presented here either implemented or to be implemented in the foreseeable future (hopefully)
Signals

- describe behavior over time
- several kinds of signals:
  - discrete values at discrete times
  - continuous values at all times
  - segmented values at all times, discrete times where changes may occur
- examples of signals:
  - discrete sequence of key presses (type $DSignal\ Char$)
  - continuous time (type $CSignal\ DiffTime$)
  - segmented title of a window (type $SSignal\ String$)
- class $Signal$ with instances for $DSignal$, $CSignal$ and $SSignal$
Circuits

- circuits for communicating with the real world
- have inputs and outputs which are tuples of signals
- kinds of circuits:
  - plain circuits
    - \texttt{timeProvider :: PlainCircuit () (CSignal DiffTime)}
  - circuits with additional “effects”
    - widget circuits
    - window circuits
- circuit composition using arrow methods
Arrows

- arrows informally:
  - effectful computations like monads
  - explicit input (contrary to monads)
  - more general than monads
- arrow syntax (simplified):
  - proc-expressions

```
proc ⟨input⟩ → do
    ⟨output₁⟩ ← ⟨arrow₁⟩ → ⟨input₁⟩
    ... ...
    ⟨outputₙ⟩ ← ⟨arrowₙ⟩ → ⟨inputₙ⟩
returnA → ⟨output⟩
```

- rec-blocks for constructing arrows with cycles
Switching and signal instantiation

- family of functions for switching between different signals

\[
switch :: (Signal \; signal) \\
\Rightarrow SS\text{signal} \; (signal \; val) \rightarrow signal \; val
\]

- different signal consumers run their own signal instance
- signal is instantiated when
  - a circuit is created which consumes the signal
  - the signal is switched into
- behavior may depend on instantiation time if signal is stateful
- memoization and early starting:

\[
with\text{Signal} \; :: \; (Signal \; signal, Signal \; signal') \\
\Rightarrow signal \; val \rightarrow (signal \; val \rightarrow signal' \; val') \\
\rightarrow signal' \; val'
\]
some discrete signal functions:

\[
\begin{align*}
\text{fmap} & : (val \rightarrow val') \rightarrow (\text{DSignal} \ val \rightarrow \text{DSignal} \ val') \\
\text{filter} & : (val \rightarrow \text{Bool}) \rightarrow (\text{DSignal} \ val \rightarrow \text{DSignal} \ val) \\
\text{scan} & : (val' \rightarrow val \rightarrow val') \rightarrow val' \\
& \rightarrow (\text{DSignal} \ val \rightarrow \text{DSignal} \ val') \\
\text{merge} & : \text{DSignal} \ val_1 \rightarrow \text{DSignal} \ val_2 \\
& \rightarrow \text{DSignal} \ (\text{MergeVal} \ val_1 \ val_2) \\
\text{data} \ \text{MergeVal} \ val_1 \ val_2 = & \ \text{First} \ val_1 \\
& | \ \text{Second} \ val_2 \\
& | \ \text{Both} \ val_1 \ val_2
\end{align*}
\]
Functions on segmented signals

- some segmented signal functions:

  \[ \text{initAndHold} :: \text{val} \rightarrow \text{DSignal val} \rightarrow \text{SSignal val} \]

  \[ \text{fmap} :: (\text{val} \rightarrow \text{val}') \rightarrow (\text{SSignal val} \rightarrow \text{SSignal val}') \]

  \[ \text{pure, return} :: \text{val} \rightarrow \text{SSignal val} \]

  \[ \text{join} :: \text{SSignal (SSignal val)} \rightarrow \text{SSignal val} \]

  \[ (\langle * \rangle) :: \text{SSignal (val} \rightarrow \text{val}') \rightarrow (\text{SSignal val} \rightarrow \text{SSignal val}') \]

  \[ (|*|) :: \text{SSignal (val} \rightarrow \text{val}') \rightarrow (\text{SSignal val} \rightarrow \text{SSignal val}') \]
Functions on continuous signals

- some continuous signal functions:

  \[
  \text{fmap} :: (\text{val} \rightarrow \text{val}') \rightarrow (\text{CSignal val} \rightarrow \text{CSignal val}')
  \]

  \[
  \text{pure} :: \text{val} \rightarrow \text{CSignal val}
  \]

  \[
  (\langle * \rangle) :: \text{CSignal (val} \rightarrow \text{val}') \rightarrow (\text{CSignal val} \rightarrow \text{CSignal val}')
  \]

  \[
  \text{withoutSegs} :: \text{SSignal val} \rightarrow \text{CSignal val}
  \]

  \[
  \text{withSegs} :: \text{DSignal ()} \rightarrow \text{CSignal val} \rightarrow \text{SSignal val}
  \]
Representing continuous sources

- If there is a way to be notified about changes (events, interrupts) than use segmented signals.
  - no polling
  - no delayed reaction
- If there is no notification mechanism than use continuous signals (even if the signal changes only at discrete points in time).
  - sampling gives the times when to read
  - user can choose among a variety of sampling schemes (clocked, sample when idle, etc.)
Records

- attribute names as first-class values

\[
data \ Caption = Caption\]
\[
data \ IsEnabled = IsEnabled\]

- records as heterogenous lists of name-value pairs whose structure is mirrored by their types
  - () for empty records
  - strict pair type using infix notation for non-empty records

\[
data \ init: & last = !init : & !last\]

- partially strict pair type for name-value pairs

\[
data \ name ::= val = !name := val\]
Records example

- example of a record:

```haskell
() :& Caption := pure "Ok"
  :& IsEnabled := fmap isValid input
```

- type of this record:

```haskell
() :& Caption :: SS Signal String
  :& IsEnabled :: SS Signal Bool
```
Advanced record features

- attribute order independence
- defaulting for input record attributes
- dropping attributes of output records
Realization of advanced record features

- in input records, name now contains optionality information

\[
() : & \text{Mandatory Caption} \quad ::= \quad \text{SSignal String}
\]
\[
: & \text{Optional} \quad \text{IsEnabled} \quad ::= \quad \text{SSignal Bool}
\]

- conversions between internal and external records (basic idea):

```haskell
class Input internalInput externalInput where
  inputConv :: externalInput → internalInput

class Output internalOutput externalOutput where
  outputConv :: internalOutput → externalOutput
```

- circuits offered by the library already contain these conversions

- input expressions and output patterns determine what instance to choose
Advanced record features example

- input type:

  \( () : & \text{Mandatory Caption} \rightarrow \text{SSignal String} \)

  \& \text{Optional IsEnabled} \rightarrow \text{SSignal Bool} \)

- output type:

  \( () : & \text{Push} \rightarrow \text{DSignal ()} \)

  \& \text{MaybeMousePos} \rightarrow \text{SSignal (Maybe Pos)} \)

- arrow statement:

  \( () : & \text{Push} := \text{ok} \leftarrow \text{button} \rightarrow () : & \text{Caption} := \text{pure "Ok"} \)
Representing toolkits

- constructorless types for denoting toolkits

  \texttt{data} GTK
  \texttt{data} Qt
  \texttt{data} NCurses
Declaring interfaces

- classes for parts of the GUI interface

```haskell
class Toolkit toolkit where
    type NativeWidget :: *
    type NativeWindow :: *

...

class (Toolkit toolkit) ⇒
    SupportsCommonWidgets toolkit where
    button :: Widget toolkit ........
    label  :: Widget toolkit ........

class (Toolkit toolkit) ⇒
    SupportsTreeView toolkit where
    treeView :: Widget toolkit ........
```
Implementing all interfaces

- complete interface supported:
  
  instance *Toolkit*  
  instance *SupportsCommonWidgets* 
  instance *SupportsTreeView*
Implementing some interfaces

- some features not supported by underlying library:
  
  \texttt{instance} \textit{Toolkit \ NCurses where} . . .
  
  \texttt{instance} \textit{SupportsCommonWidgets \ NCurses where} . . .
  
  -- no support for tree views in ncurses

- some features not supported because of porter’s lazyness:
  
  \texttt{instance} \textit{Toolkit \ Qt where} . . .
  
  \texttt{instance} \textit{SupportsCommonWidgets \ Qt where} . . .
  
  -- tree views not yet ported
Advantages over using linking tricks etc.

- usage of multiple toolkits in same application/GHCi session
- no extra tools or compiler options needed for toolkit selection
- dependencies of feature sets can be declared (via subclassing)
- required compliance level encoded in the types
  - compliance level can be checked
  - required compliance level can be inferred
Declarative Programming of Interactive Systems with Grapefruit

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Software Technology Colloquium at Utrecht Universiteit
May 29, 2008
Arrow composition

first

loop